Tutorial: Optimizing Applications for Performance on POWER

Application Optimization

SCICOMP13

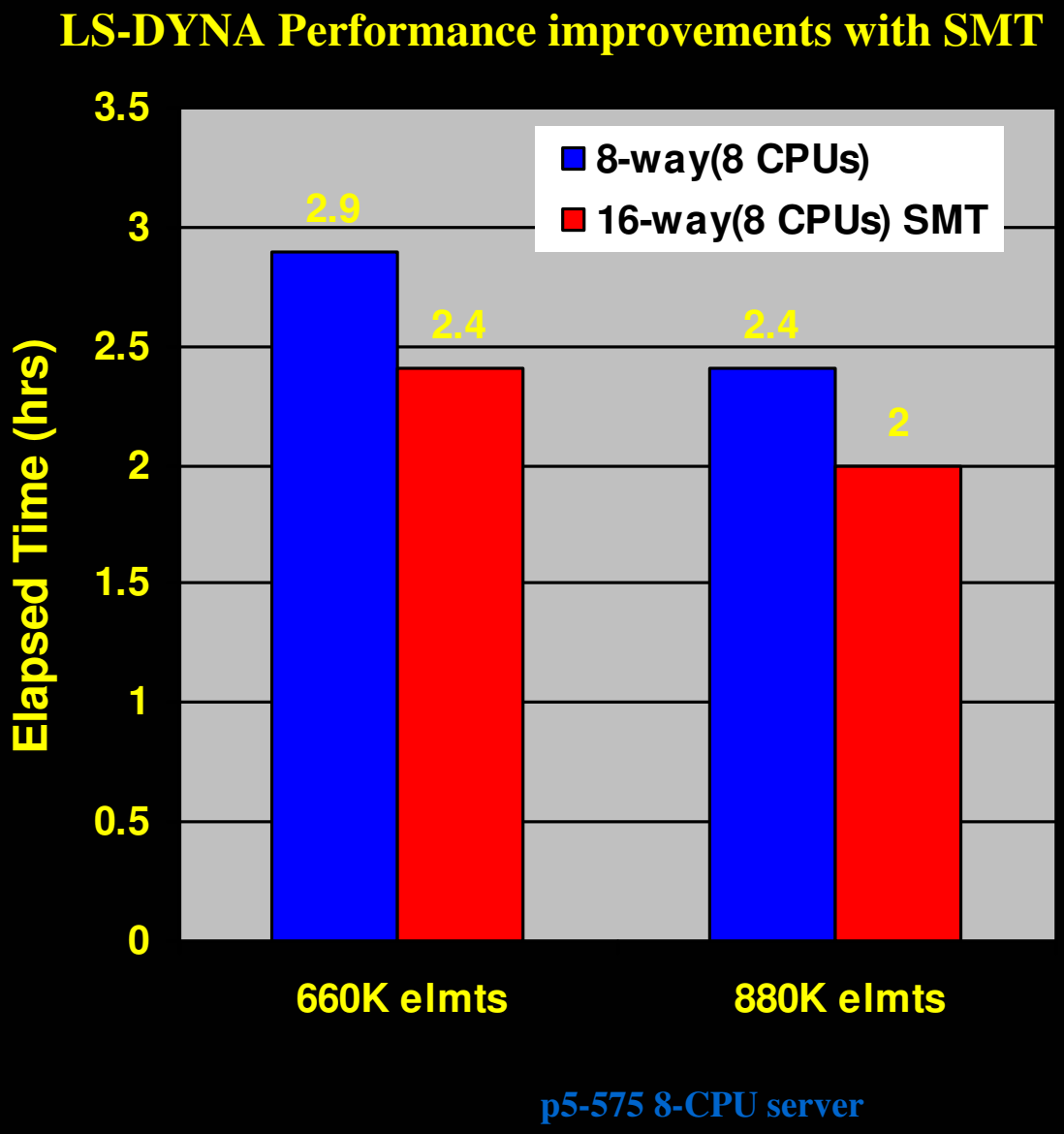
July 2007

Agenda

- System tuning
  - SMT
  - Memory
- Application tuning
  - Loop unrolling
  - Striding, blocking
  - Memory
  - Precision, address mode
  - Functions
  - Libraries
  - POWER6

Using SMT

- SMT controlled by /usr/bin/smtctl
- Can be useful when programs are not CPU bound
- Typically used with MPI codes by assigning mpi task to each virtual thread
- Care must be taken when not using all virtual threads
  - task binding keeps threads bound to real cores

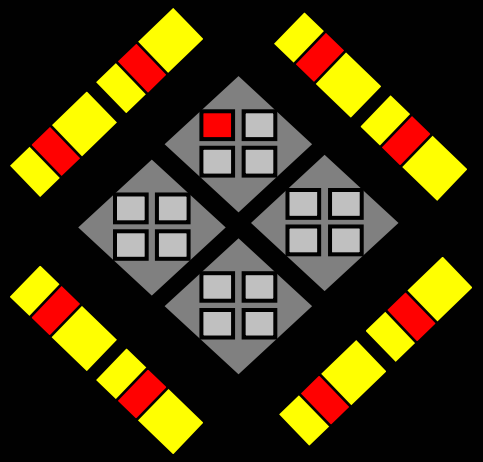


Memory Page Placement

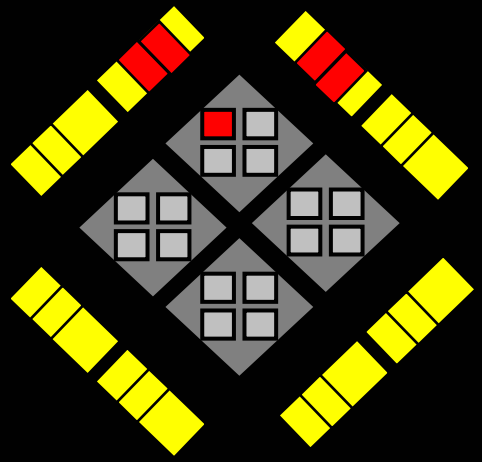
- Default is random page placement
  - Small pages
- Local page placement optional with "first touch" policy
- "Round robin" option available with AIX 5.2
- Large pages are also available
  - Loader option or "tag" binary
  - Large pages are statically allocated
  - Placed at allocation time, not first reference

Memory Allocation

- Default
  - Pages are allocated by module
  - Approximately uniform distribution
  - Approximately round robin
- Optional
  - "First Touch"
    - *setenv MEMORY_AFFINITY MCM*
    - Useful for MPI programs



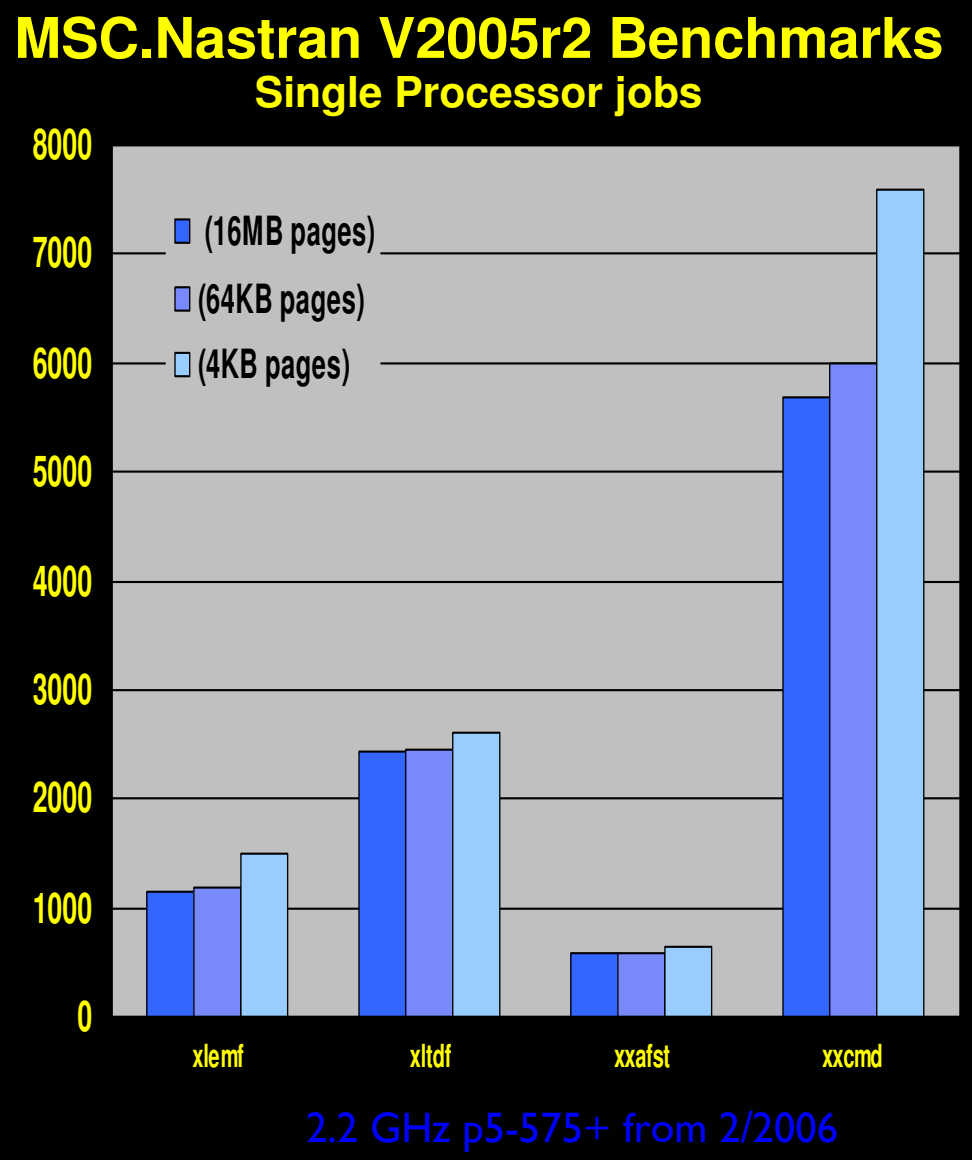
round robin



first touch

Effect of Page Size on Application Performance

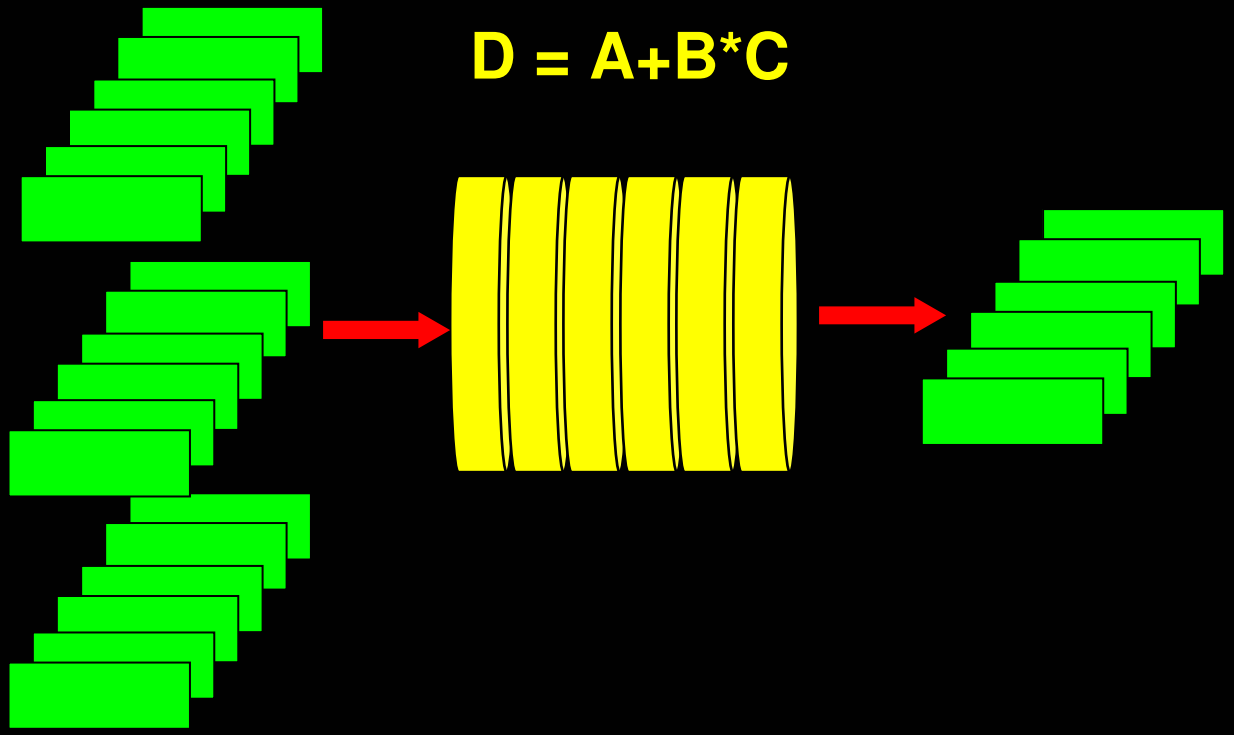
- Large pages available on POWER4, etc
- Medium pages available on POWER5+, etc
- Enable large pages with executable
  - ldedit –blpdata a.out
- Enable medium pages
  - ldedit -bdatapsize=64K a.out
  - ldedit -btextpsize=64K a.out
  - ldedit -bstackpsize=64K a.out
- Changing data pages seems to have the largest performance effect



Strategies for Optimization

- Use optimized libraries
- Exploit multiple functional units
- Exploit FMA capability
- Pipelined operations
- Cache reuse
  - Blocking
- Unit stride
  - Use entire loaded cache lines
- Limit range of indirect addressing
  - Sort indirect addresses
- Increase computational intensity of the loops
  - Ratio of flops computed to bytes moved
- Expose load/store “streams”
  - utilize hardware prefetching

FMA Functional Units



- Multiply and Add
  - Accuracy is better than separate multiply and add
  - may need to compile -qnofma to compare with other results
- 2 FMA units per core
  - 1 result per clock
  - 2 flt pt ops/clock * 2 units / core * 2.2e09 clocks/sec = 8.8 Gflops / core
  - 6 clock period latency

Programming Concerns

- Superscalar design
  - Concurrency:
    - Branches
    - Loop control
    - Procedure calls
    - Nested “if” statements
  - Program “control” is very efficient
- Cache based microprocessor
  - Critical resource: memory bandwidth
    - Tactics:
      - Increase Computational Intensity
      - Exploit "prefetch"

Computational Intensity

- Ratio of floating point operations per memory references (loads and stores)
- Higher is better
- Example:

```
for (i=0;i<n;i++)
  A[i] = A[i] +B[i]*s+C[i];
```

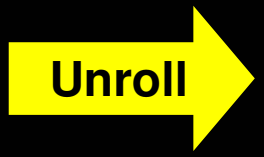
  - Loads and stores: 3+1
  - Floating point operations: 3
  - Computational intensity: 3/4

Loop Unrolling Strategy:
Increase Computational Intensity

- Find variable which is constant with respect to outer loop
  - Unroll such that this variable is loaded once but used multiple times
- Unroll outer loop
  - Minimizes load/stores

Outer Loop Unroll

```
DO I = 1, N
     DO J = 1, N
        s =  s +X(J)*A(J,I)
     END DO
END DO
```



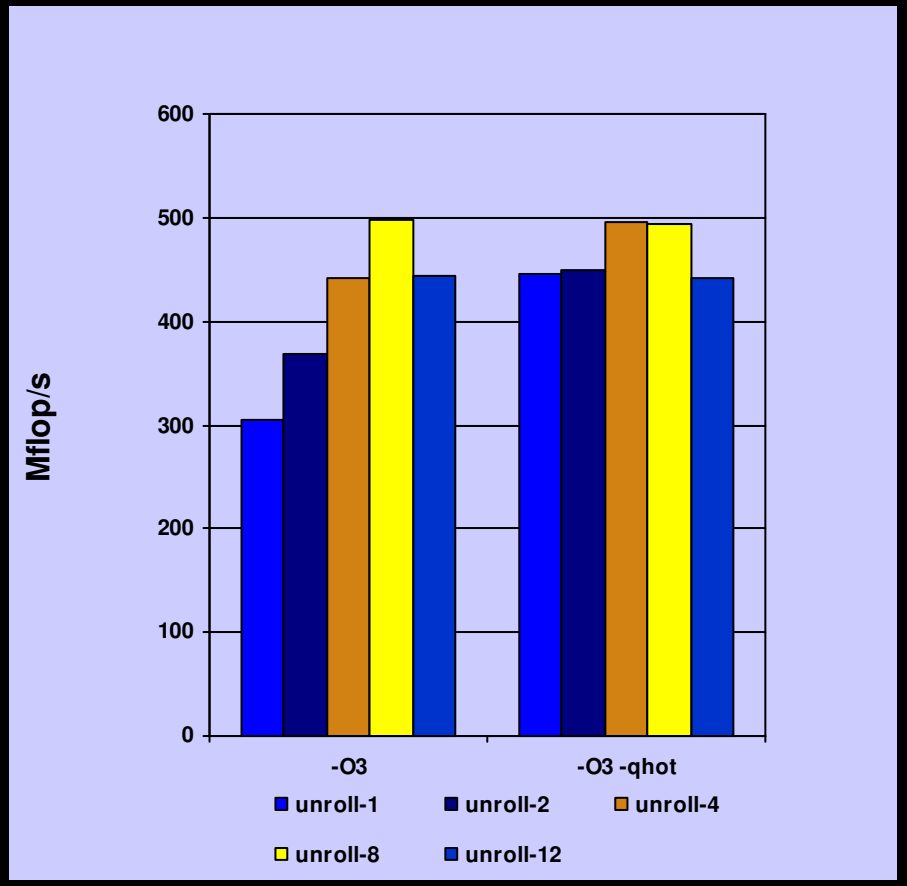

```
DO I= 1, N, 4
  DO J = 1, N
     s = s +X(J)*A(J,I+0)
           +X(J)*A(J,I+1)
           +X(J)*A(J,I+2)
           +X(J)*A(J,I+3)
  END DO
END DO
```

- 2 flops / 2 loads
- Comp. Int.: 1

- 8 flops / 5 Loads
- Comp. Int.: 1.6

Outer Loop Unroll Test

- Strategy:
  - Unroll up to 8 times
    - Exploit all 8 prefetch streams
- Compiler (XL Fortran 8.1):
  - Unrolls up to 4 times
    - "Near" optimal performance
  - Combines inner and outer loop unrolling



POWER4 1.3 GHz

Outer Loop Unrolling Strategies

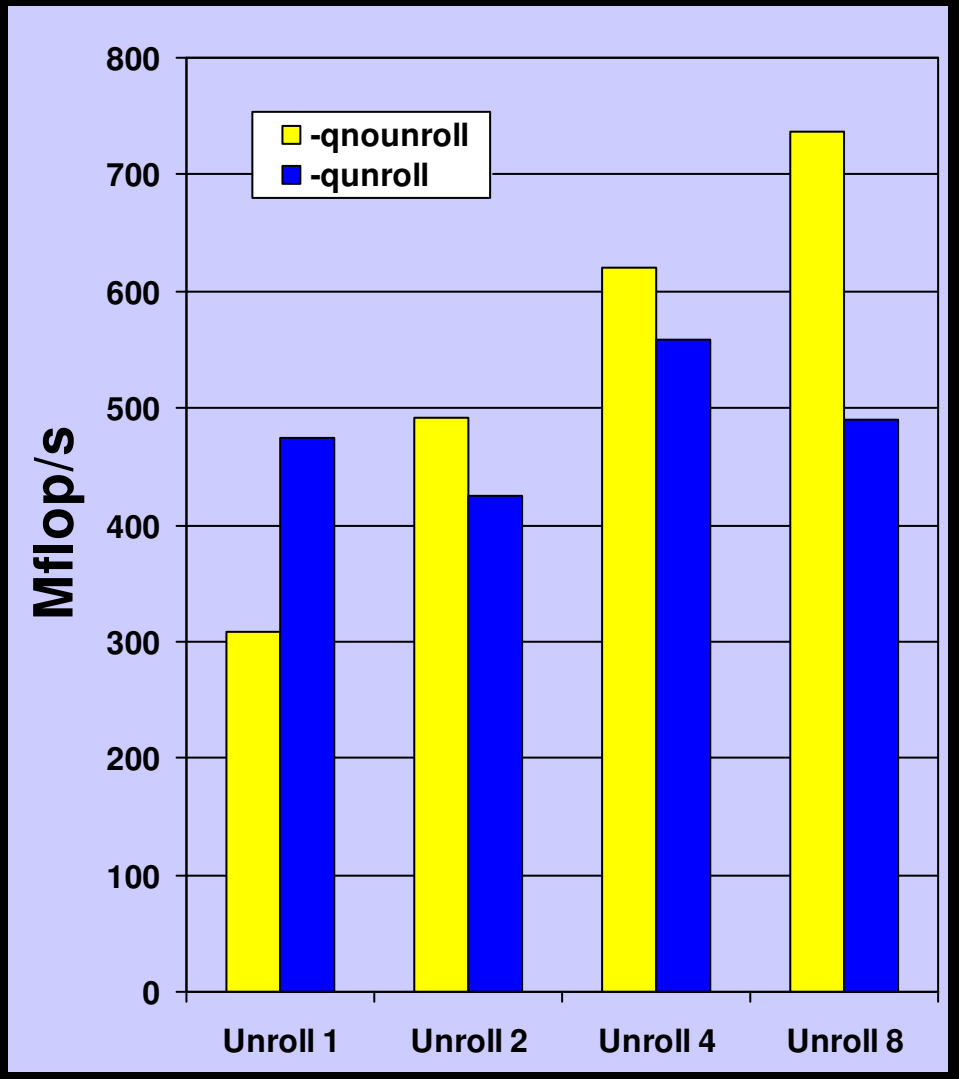
- Expose prefetch streams
  - Up to 8 streams

```
do j=1,n
    do i=1,m
        sum = sum + X(i)*A(i,j)
    end do
end do
```

```
do j=1,n,2
   do i=1,m
       sum = sum + X(i)*A(i,j)    &
                 + X(i)*A(i,j+1)
   end do
end do
```

Outer Loop Unroll: Streams

- Compiler does a “fair” job of outer loop unrolling
- Manual unrolling can outperform compiler unrolling



POWER4 1.45 GHz

Loop Unrolling Strategies

- Inner loop strategy:
  - Reduces data dependency
  - Eliminate intermediate loads and stores
  - Expose functional units
  - Expose registers
  - Examples:
    - Linear recurrences
    - Simple loops
      - Single operation

Inner Loop Unroll: Dependencies

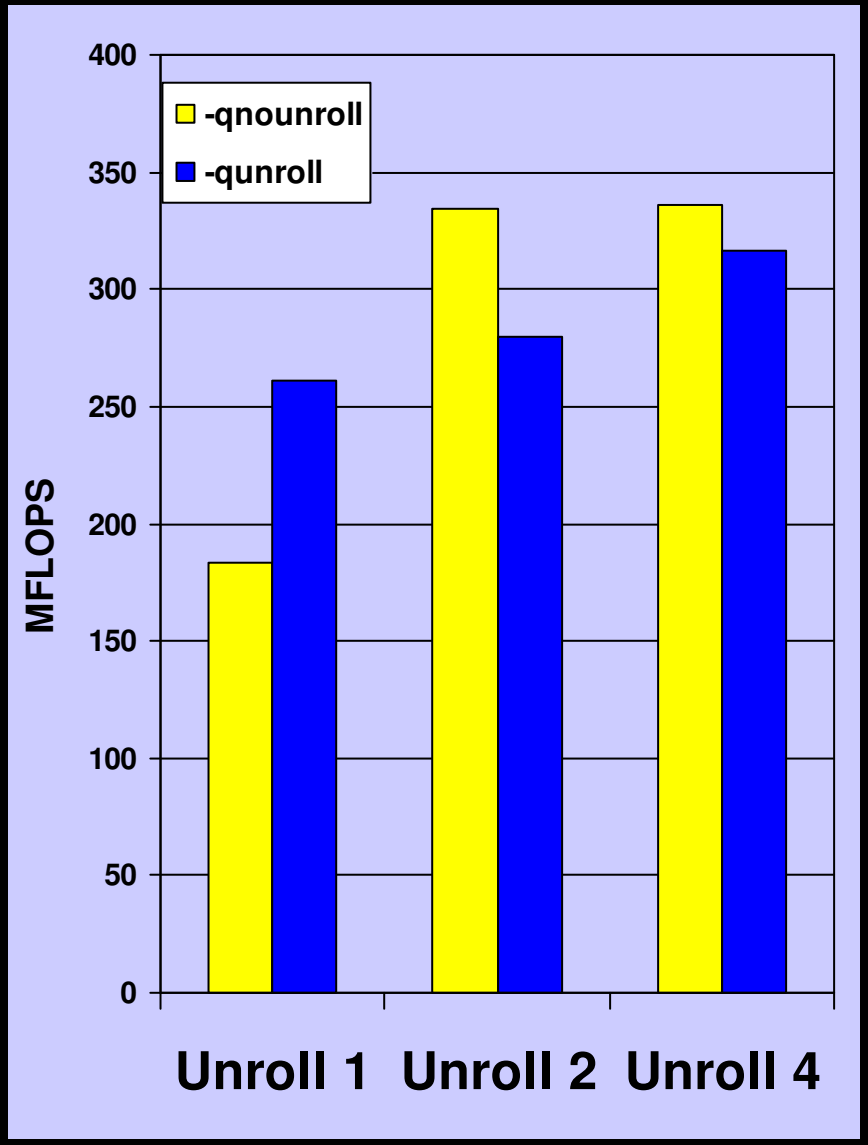
- Eliminate data dependence (half)
- Eliminate intermediate loads and stores
  - Compiler will usually do this at -O3 and higher

```
do i=2,n-1
   a(i+1) = a(i)*s1 + a(i-1)*s2
end do
```

```
do i=2,n-2,2
     a(i+1) = a(i  )*s1 + a(i-1)*s2
     a(i+2) = a(i+1)*s1 + a(i  )*s2
 end do
```

Inner Loop Unroll: Dependencies

- Compiler unrolling helps
  - But, does not help manually unrolled loops
- Turn off compiler unrolling if manually unrolled



POWER4 1.45 GHz

Inner Loop Unroll: Registers and Functional Units

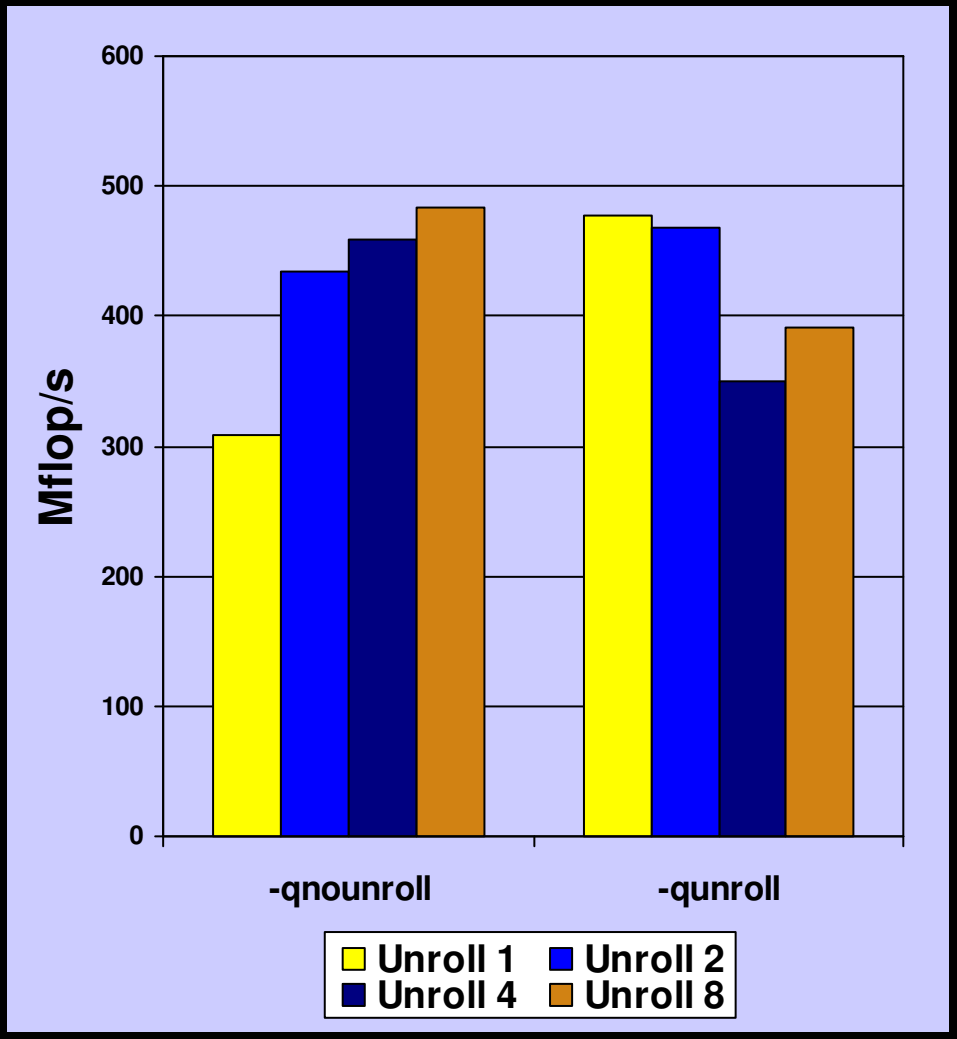
- Expose functional units
- Expose registers
  - Compiler will do some of this at -O3 and higher

```
do j=1,n
     do i=1,m
          sum = sum + X(i)*A(i,j)
     end do
end do
```

```
do j=1,n
   do i=1,m,2
     sum = sum + X(i)   *A(i,j)    &
               + X(i+1)*A(i+1,j)
   end do
end do
```

Inner Loop Unroll: Registers and Functional Units

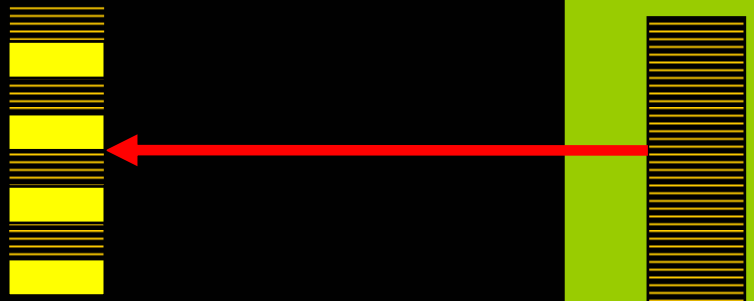
- Compiler does adequate job of unrolling
- Do not manually unroll inner loop
- Turn off unroll is code is manually unrolled

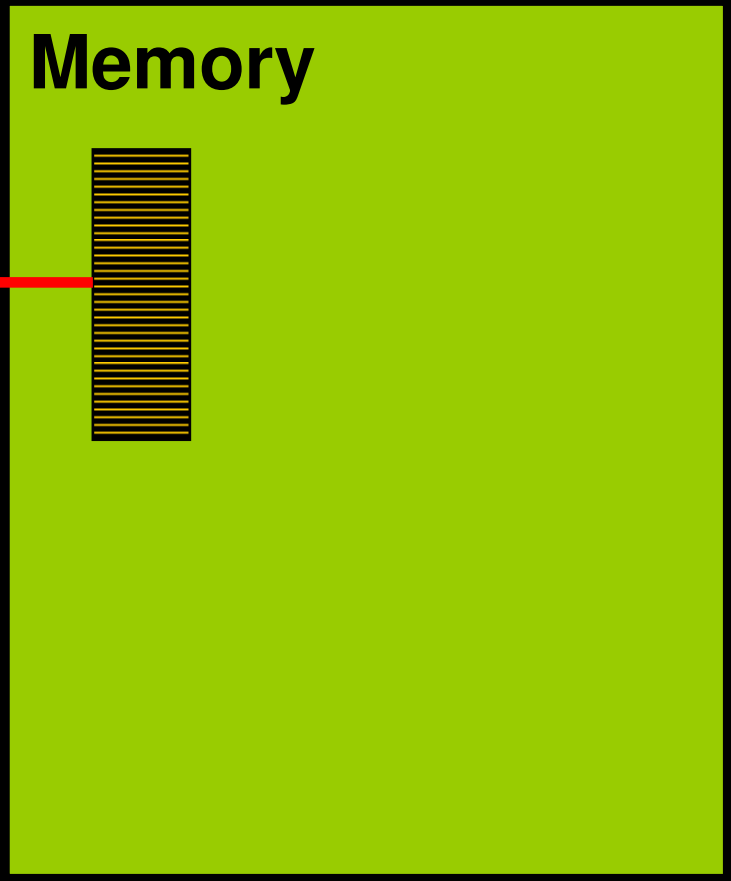


POWER4 1.45 GHz

Strides: Cache Lines

```
for (i=0;i<n;i+=2)
    sum+=a[i];
```





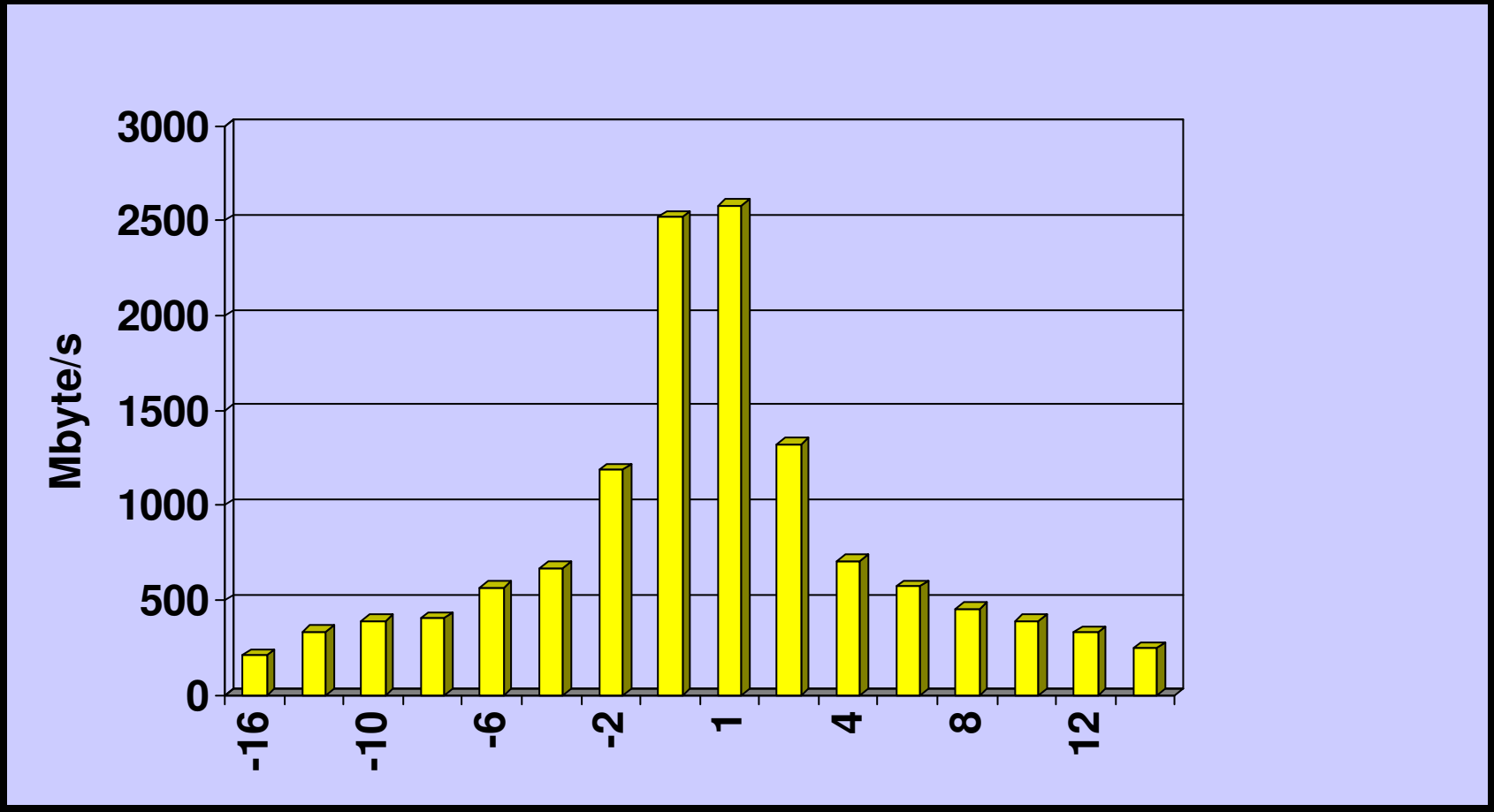
- Strided memory accesses use partial cache lines
  - Reduced efficiency

Strides

- Cache line size is 128 bytes
  - Double precision: 16 words
  - Single precision: 32 words

	Bandwidth Reduction	
Stride	Single	Double
1	1x	1x
2	½	½
4	¼	¼
8	1/8	1/8
16	1/16	1/16
32	1/32	1/16
64	1/32	1/16

Stride Test



POWER4 1.3 GHz

Correcting Strides

- Interleave code
- Example:
  - Real and Imaginary part arithmetic

```
do i=1,n
    … AIMAG(Z(i))
end do
do i=1,n
    …REAL(Z(i))
end do
```

```
do i=1,n
    … AIMAG(Z(i))
…
    …REAL(Z(i))
end do
```

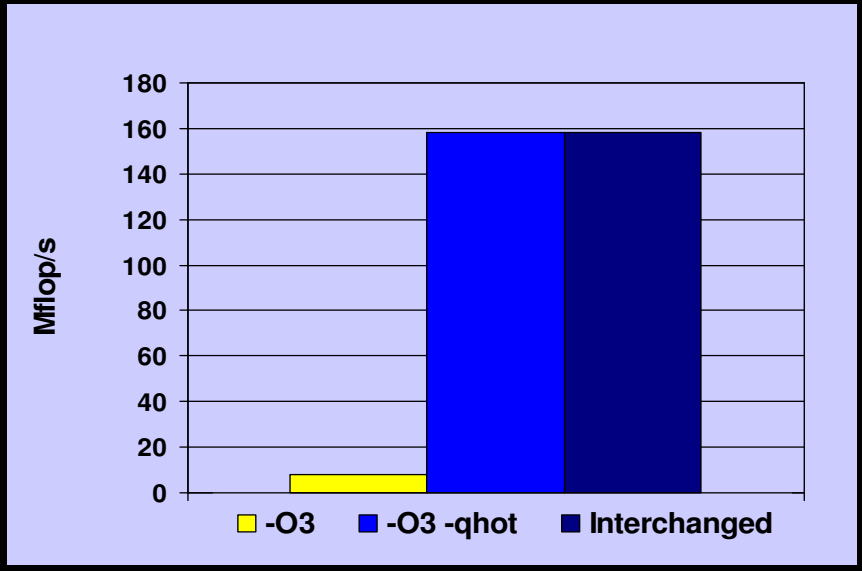
Correcting Strides

interchanging loops

```
do i=1,n
   do j=1,m
      sum=sum+A(i,j)
    end do
end do
```

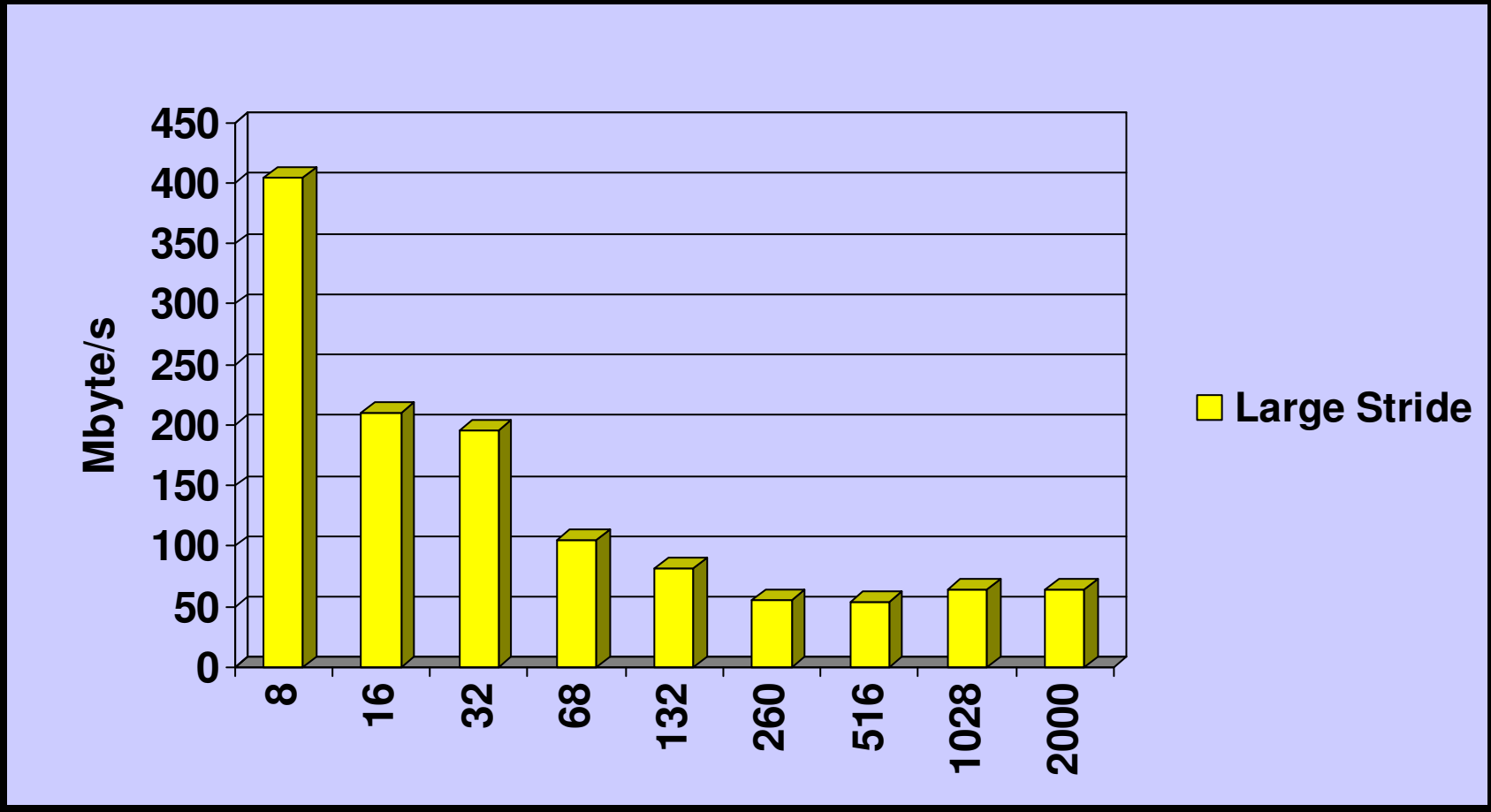
Interchange

```
do j=1,m
   do i=1,n
      sum=sum+A(i,j)
    end do
end do
```



POWER4 1.45

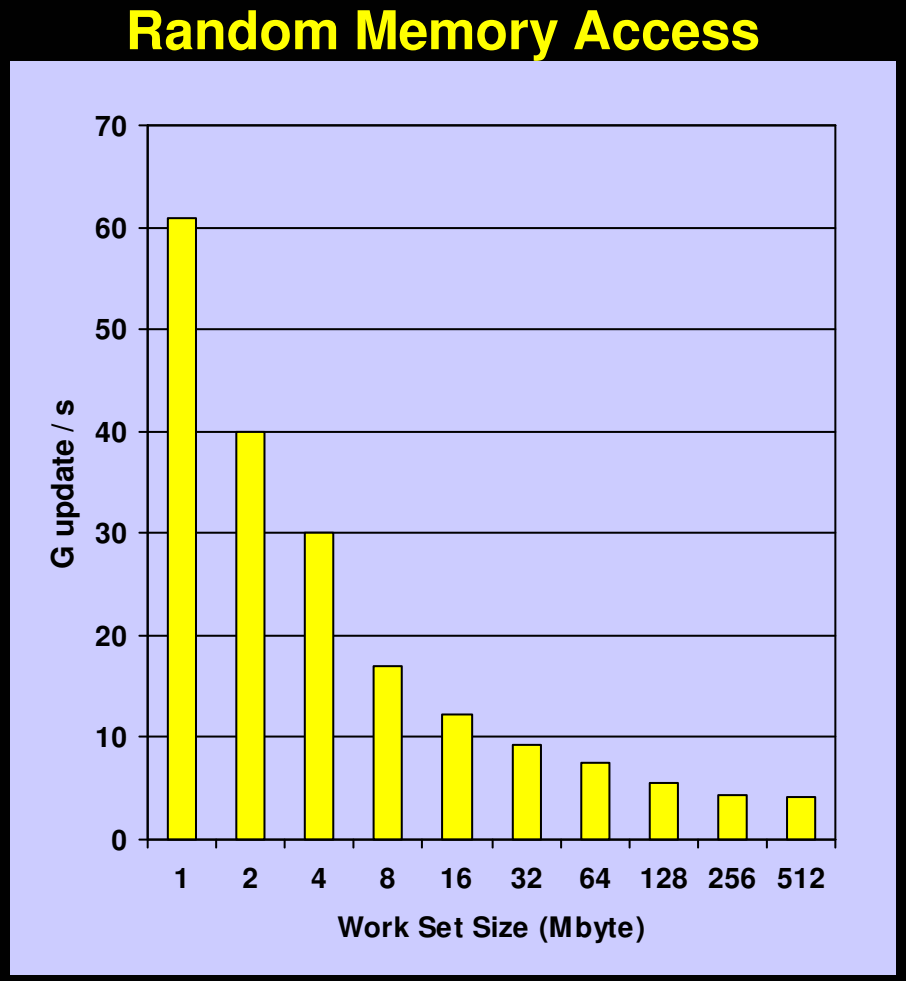
Effect of Large Strides (TLB Misses)



POWER4 1.3 GHz

Working Set Size

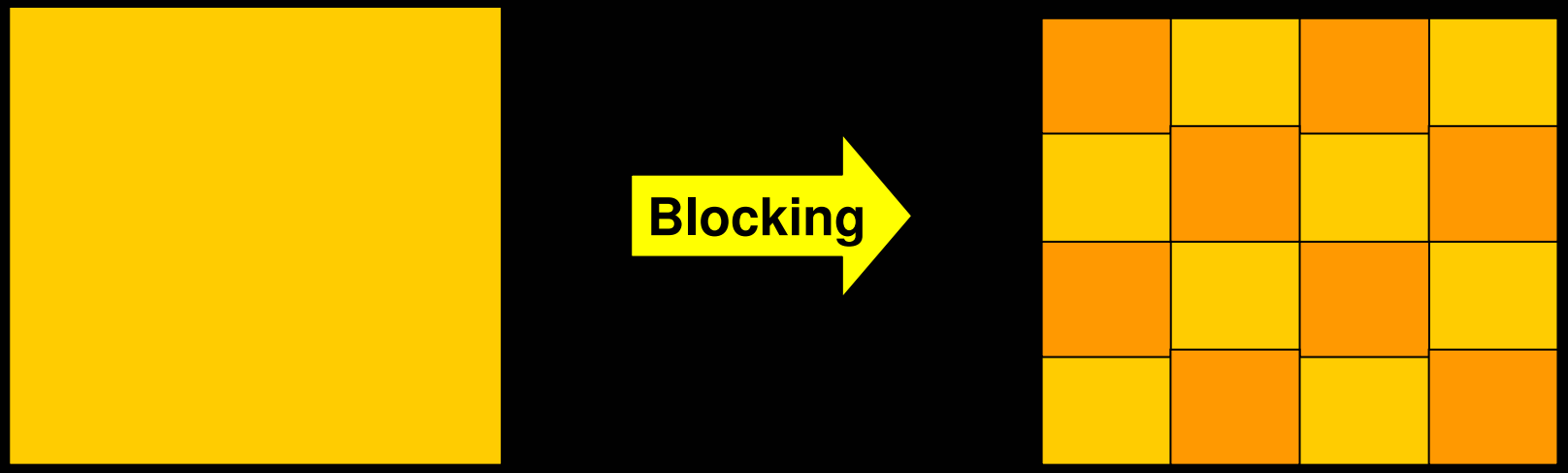
- Reduce spanning size of random memory accesses
- More MPI tasks usually helps



POWER4 1.3 GHz

Blocking

- Common technique in Linear Algebra (LA)
  - Similar to unrolling
  - Utilize cache lines
  - Linear Algebra NB:
    - Typically 96-256

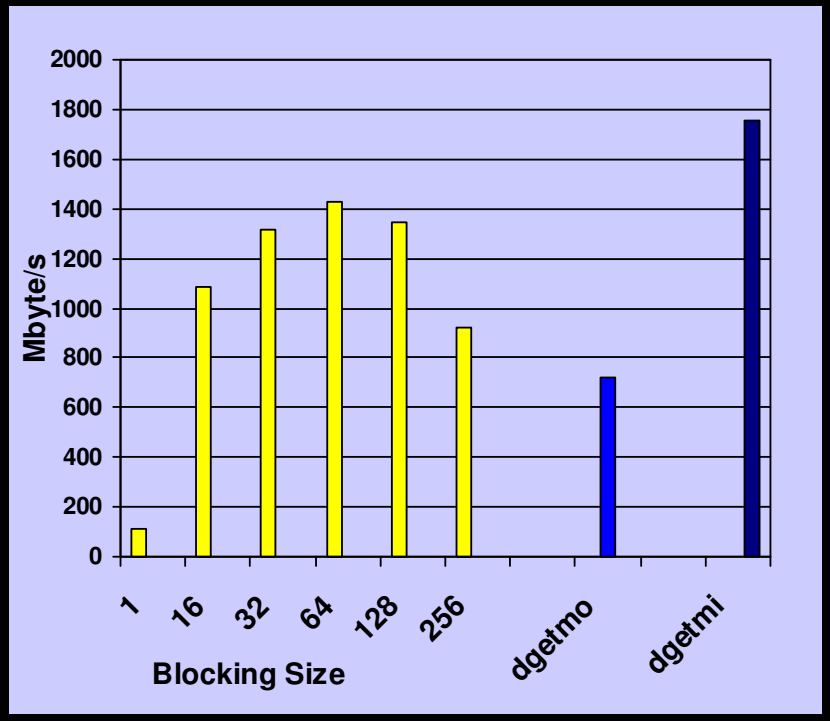


Blocking Example: Transpose

```
do i = 1,n
  do j = 1,m
    B(j,i) = A(i,j)
  end do
end do
```

Blocking

```
do j1 = 1,n-nb+1,nb
   j2 = min(j1+nb-1,n)
   do i1 = 1,m-nb+1,nb
      i2 = min(i1+nb-1,m)
      do i = i1, i2
        do j = j1, j2
           B(j,i) = A(i,j)
        end do
      end do
   end do
end do
```



- Especially useful for bad strides

Hardware Prefetch

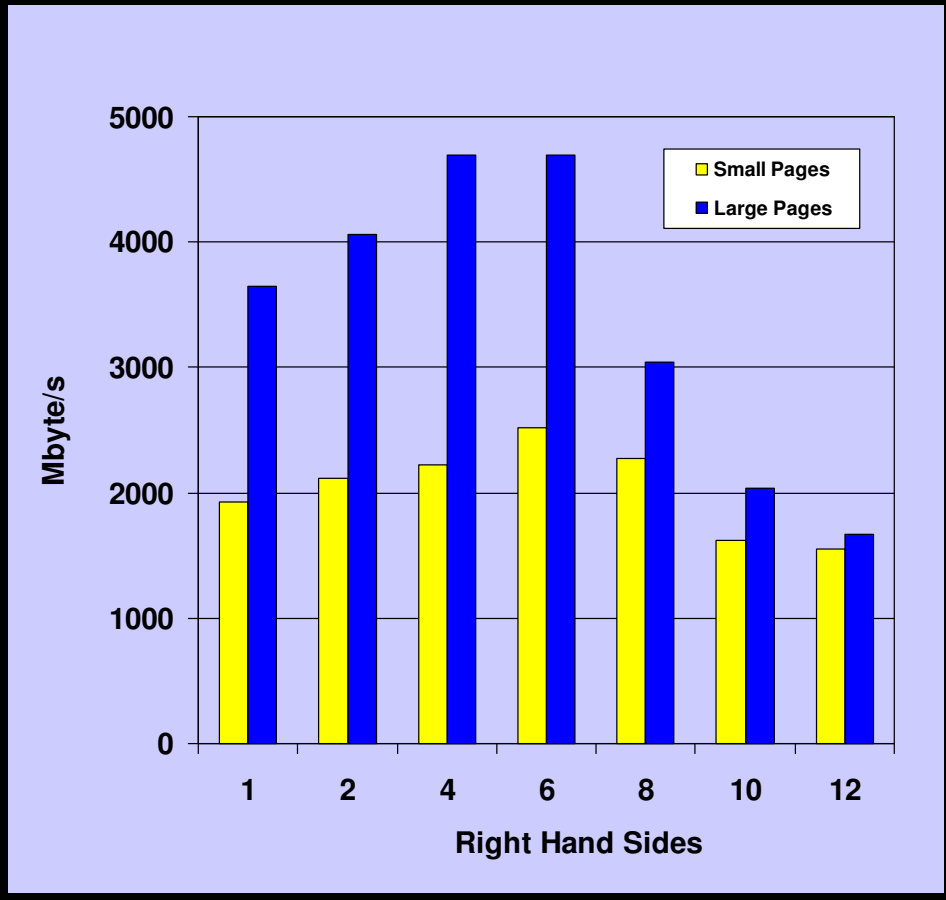
- Detects adjacent cache line references
- Forward and backward
- Up to eight concurrent streams
- Prefetches up to two lines ahead per stream
- Twelve prefetch filter queues prevents rolling
- No prefetch on store misses
  - (when a store instruction causes a cache line miss)

Prefetch: Stride Pattern Recognition

- Upon a cache miss:
- Biased guess is made as to the direction of that stream
- Guess is based upon where in the cache line the address associated with that miss occurred
- If it is in the first 3/4, then the direction is guessed as ascending
- If in the last 1/4, the direction is guessed descending

Memory Bandwidth

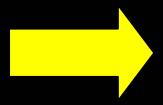
- Bandwidth is proportional to number of streams
  - Streams are roughly the number of right hand side arrays
  - Up to eight streams



POWER4 1.3 GHz

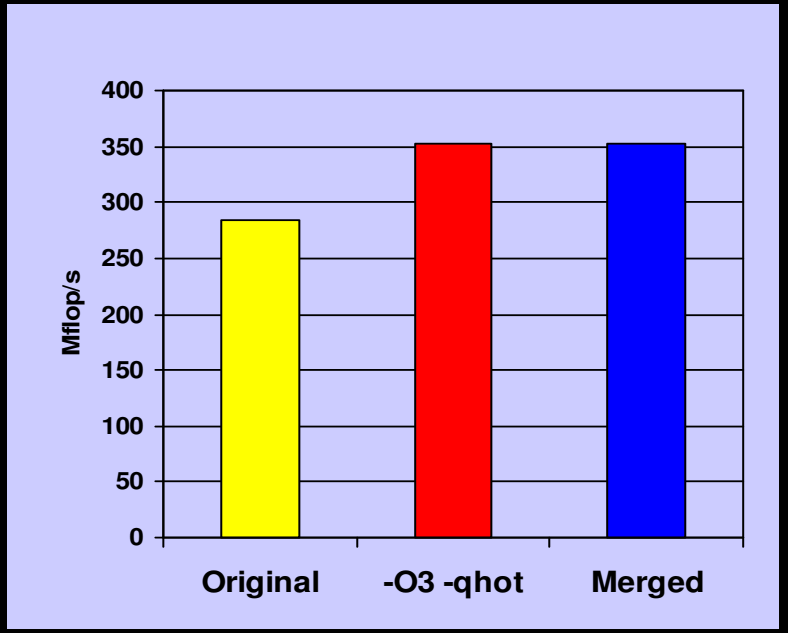
Exploiting Prefetch

```
for (j=1; j<= n; j++)
        A[j] = A[j-1]+B[j];
...
for (j=1; j<= n; j++)
        D[j] = D[j+1]+C[j]*s;
```



```
for (j=1; j<= n; j++)
        {
        A[j] = A[j-1]+B[j];
        D[j] = D[j+1]+C[j]*s;
        }
```

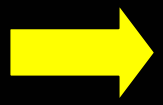
- **Strategy:**
  - merge loops to get up to 8 right hand sides
  - Compiler is able to automatically merge with "-O3 –qhot"



POWER4 1.3 GHz

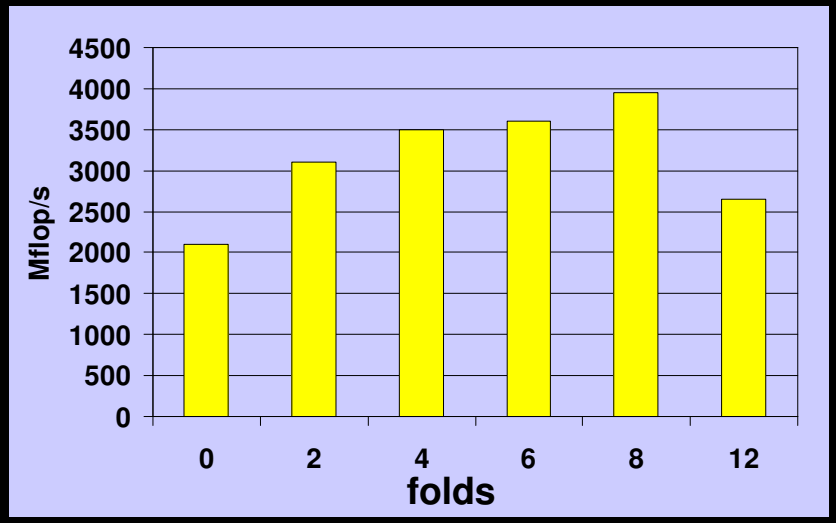
Folding

```
do i = 1,n
   sum = sum +A(i)
end do
```



```
do i = 1,n/4
sum = sum + A(i )
          + A(i+1*n/4)
          + A(i+2*n/4)
          + A(i+3*n/4)
end do
```

- **Strategy:**
  - **Fold loop to increase number of streams**



POWER4 1.3 GHz

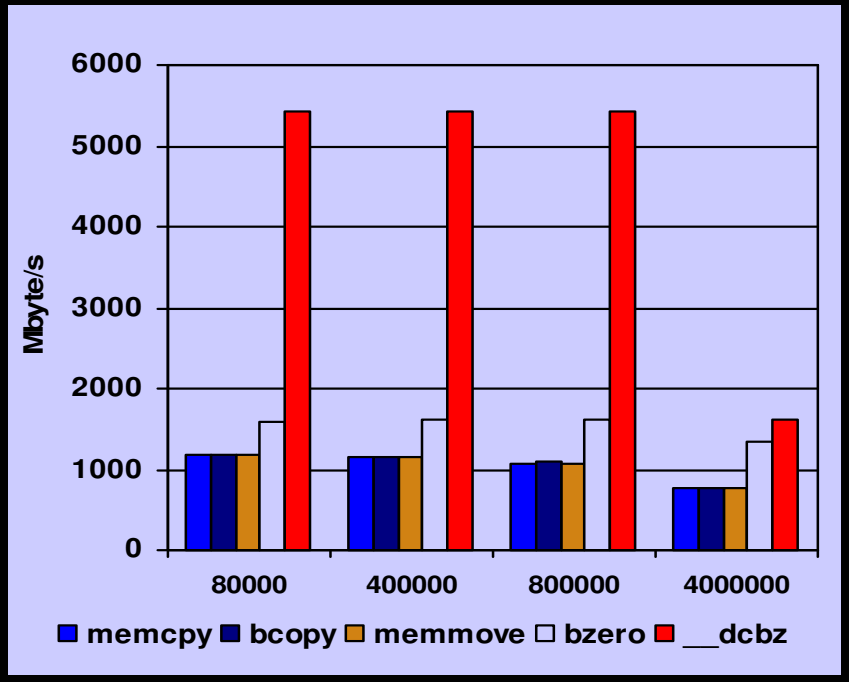
Memory Store

- Memory stores are (generally) significantly slower than memory loads
  - Avoid stores ☺
  - POWER5 2x better than POWER4
  - Accumulate on small arrays
    - Register
    - L1 or L2 cache
  - DCBZ instruction

DCBZ

- DCBZ: Data Cache Block set to Zero
  - C intrinsic:
    - __dcbz(void *);
- Write to cache line bypassing cache coherency

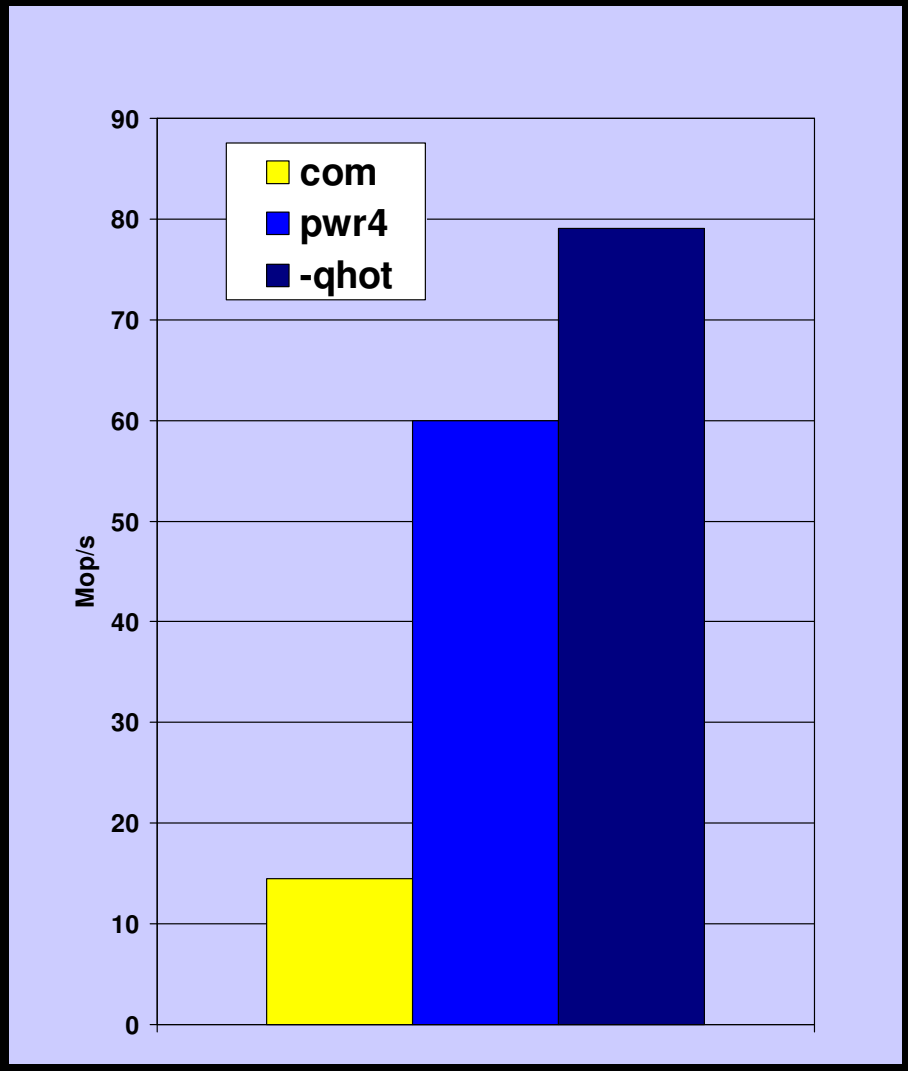
```
#define  CACHE_LINE 128
…
#if (defined(_ARCH_PWR4) || defined(_ARCH_PWR5))
    n_end=(n-n%(CACHE_LINE/sizeof(B[0])));
/*    for (i=0;i<n_end;i+=16) */
    for (i=0;i<n_end;i+=(CACHE_LINE/sizeof(B[0]))
      __dcbz(&B[i]);
/* partial cache line at end */
    for (i=n_end;i<n;i++)
       B[i]=0;
#else
    for (i=0;i<n;i++)
       B[i]=0;
#endif
```



POWER5 1.65 GHz

SQRT

- POWER4/POWER5 have hardware SQRT available
  - Default -qarch=com uses software library
  - Use: -qarch=pwr4,pwr5
- Hardware SQRT is 5x faster
- -qhot generates "vector sqrt"

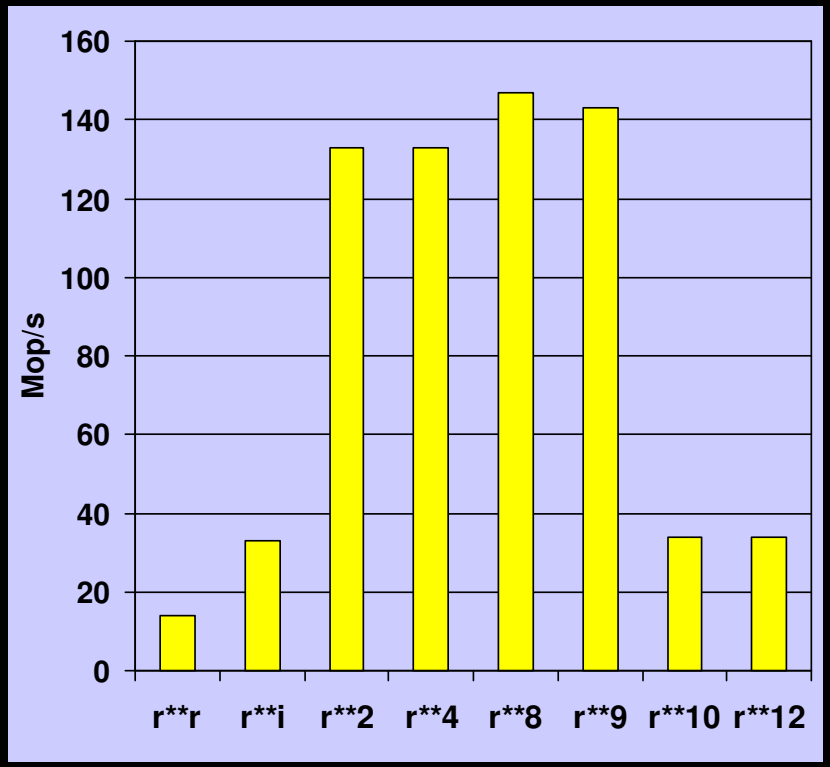


Power Function

- Computing power function: a**b
  - if (b < 10 and has integer value at compile time)
    - Use unrolling and successive multiply
  - else
    - ... __pow(...)
- Compiler can transform integer power to multiply's
- Real to Real is expensive
- Real to Integer is less expensive

Test case:

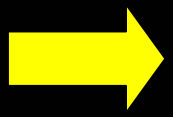
```
do i=1,n
   A(i)=B(i)**b
end do
```



POWER4 1.3 GHz

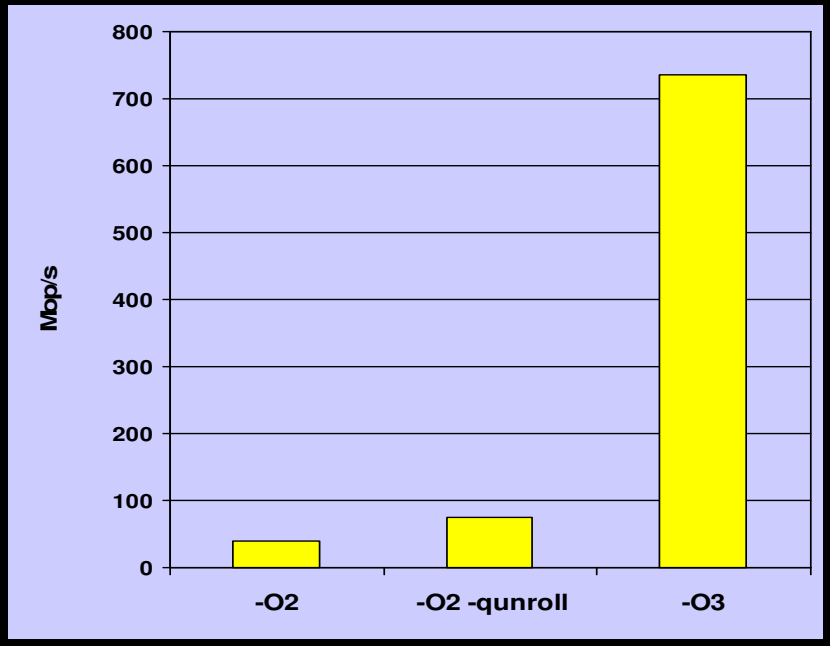
Divide

```
do i=1,n
   B(i) = A(i)/s
end do
```



```
rs=1/s
do i=1,n
   B(i) = A(i)*rs
end do
```

- IEEE divide specifies actual divide
  - Does not use multiply by reciprocal (default)
  - Optimize with -O3



POWER4 1.3 GHz

Effect of Precision

- Available floating point formats:
  - real (kind=4)
  - real (kind=8)
  - real (kind=16)
- Advantage of smaller data types:
  - Hardware speed is same for 32bit and 64bit
  - Require less bandwidth
  - More effective cache use

```
REAL*8 A,pi,e
…
do i=1,n
  A(i) = pi*A(i) + e
end do
```

```
REAL*4 A,pi,e
…
do i=1,n
  A(i) = pi*A(i) + e
end do
```

Address Mode: -q{32,64}

- Available application modes:
  - -q32
    - Default
  - -q64
  - Also: environment variable OBJECT_MODE
    - export OBJECT_MODE={32,64}
  - Cannot mix -q32 objects with -q64 objects
- AIX kernel modes:
  - 32-bit
  - 64-bit
- Application's address mode is independent of AIX

Even more on 64-bit...
(because it is so often confused)

- 64-bit floating point representation is higher precision
  - Fortran: REAL*8, DOUBLE PRECISION
  - C/C++: double
  - You can use 64-bit floats with –q32 or –q64

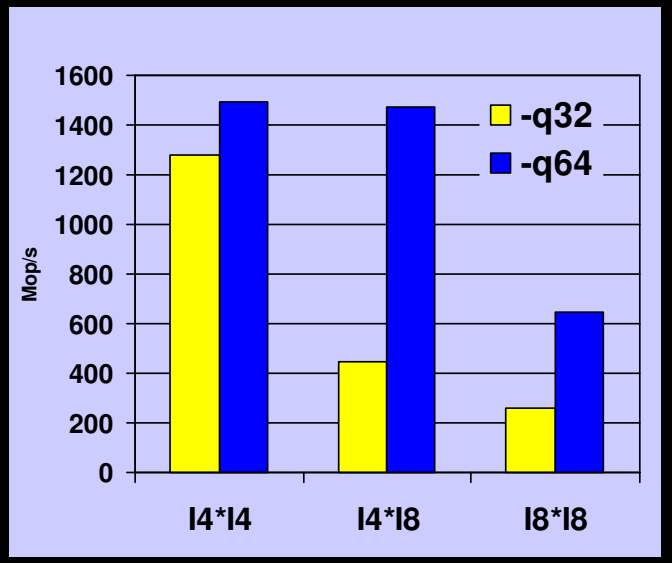
- 64-bit addressing is totally different. It refers to how many bits are used to store memory addresses and ultimately how much memory one can access.
  - Compile and link with –q64
  - Use `file a.out myobj.o` to query addressing mode

- The AIX kernel can be either a build that uses 32-bit addressing for kernel operations or uses 64-bit addressing but that does not affect an application's addressabilty.
  - ls –l /unix
  - Certain system limits depend on kernel chosen.

Effect of 32- and 64- bit Addressing

- 64-bit address mode enable use of 64-bit integer arithmetic
- Integer Arithmetic, especially (kind=8), is much faster with -q64

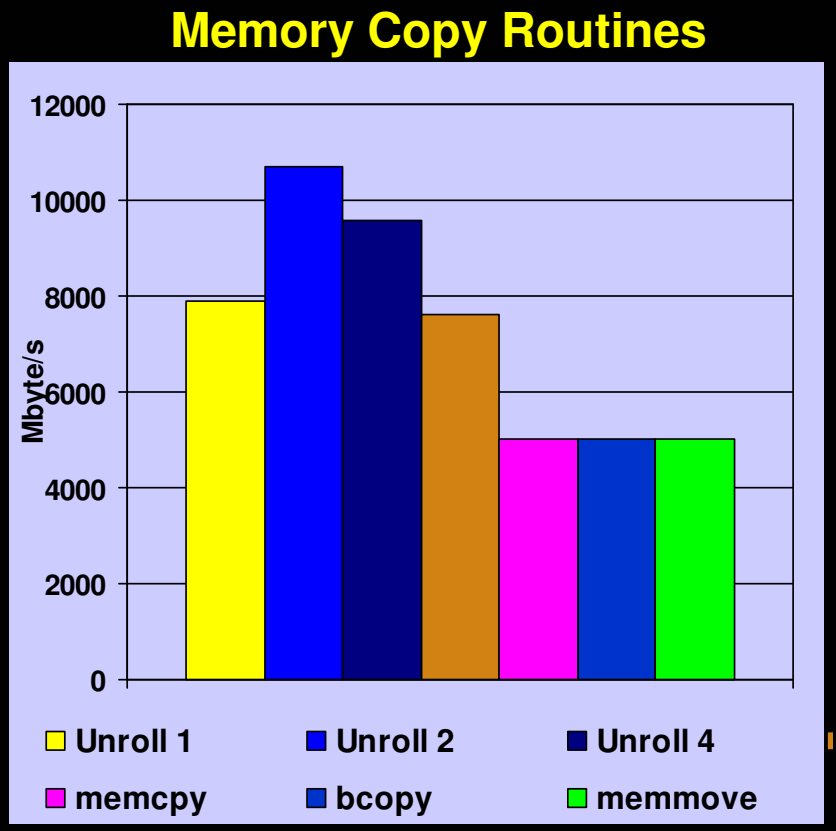
		Fortran		C, C++	
Address Mode	Computation	Integer *4	Integer *8	long	long long
-q32	4 bytes	4 bytes	8 bytes	4 bytes	8 bytes
-q64	4 or 8 bytes	4 bytes	8 bytes	8 bytes	8 bytes



POWER4 1.3 GHz

Use of Library Routines:
Level 1

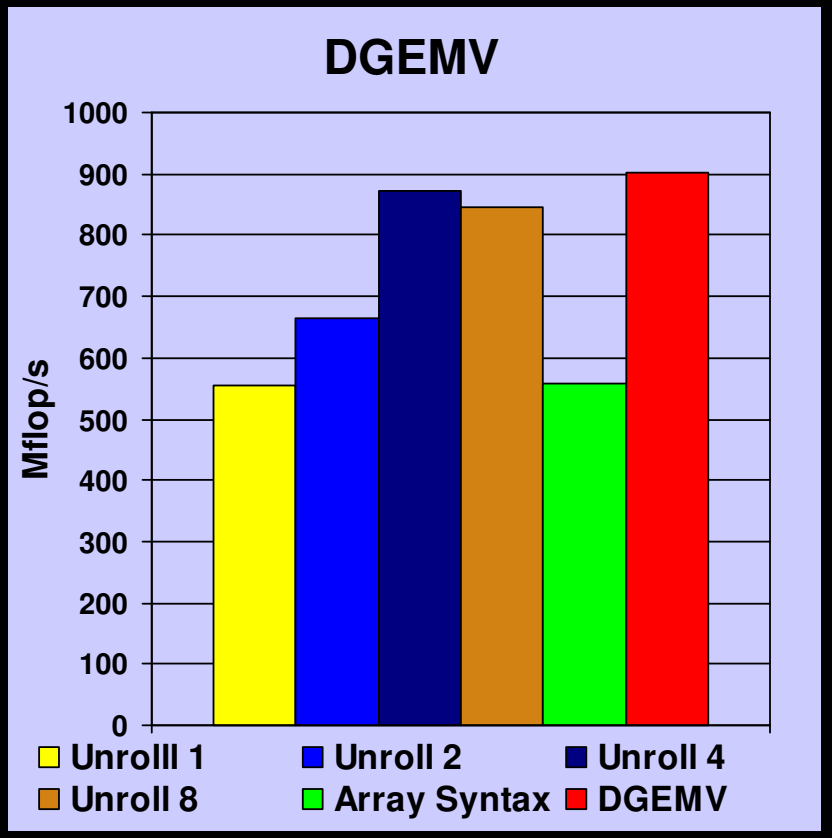
- Single level loop constructs
  - Examples
    - BLAS 1
    - Memory Copy
  - Prefer compiler generated code
    - Better than:
      - libc.a
        - memcopy
        - bcopy
        - memmove
      - ESSL
        - dcopy



POWER4 1.3 GHz

Use of Library Routines:
Level 2

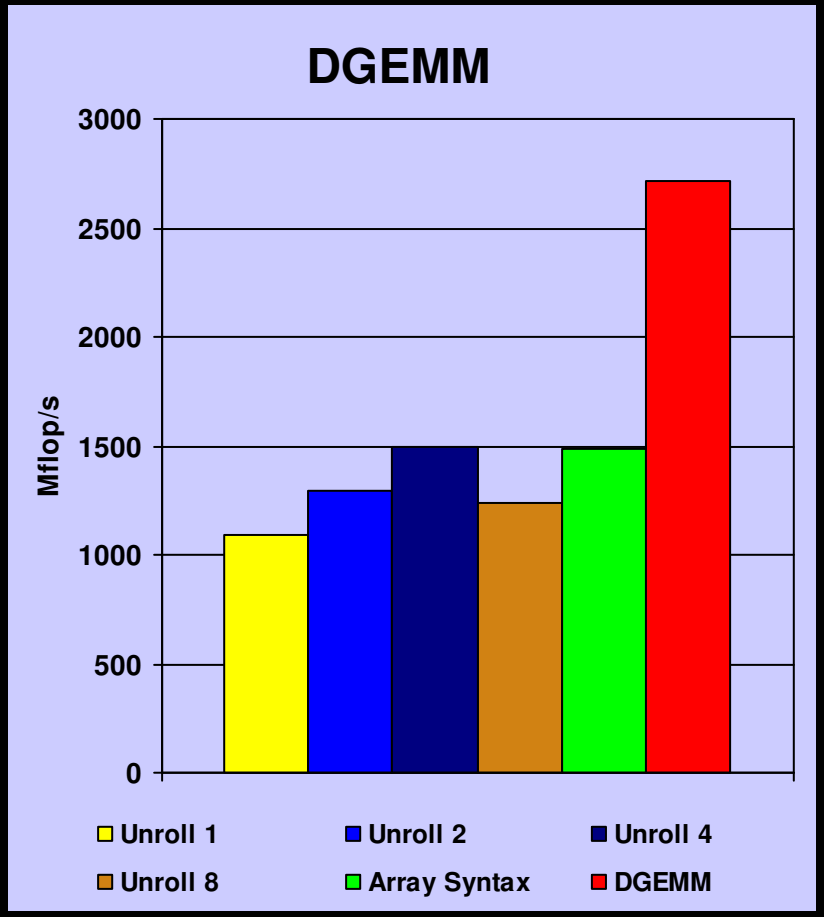
- Two level loop constructs
- Examples
  - Matrix Vector Multiply
    - DGEMV
- Prefer ESSL or liblapack
- New compilers may be better



POWER4 1.3 GHz

Use of Library Routines: Level 3

- Three level loop constructs
- Examples
  - Matrix Matrix Multiply
  - DGEMM
- Prefer ESSL or liblapack

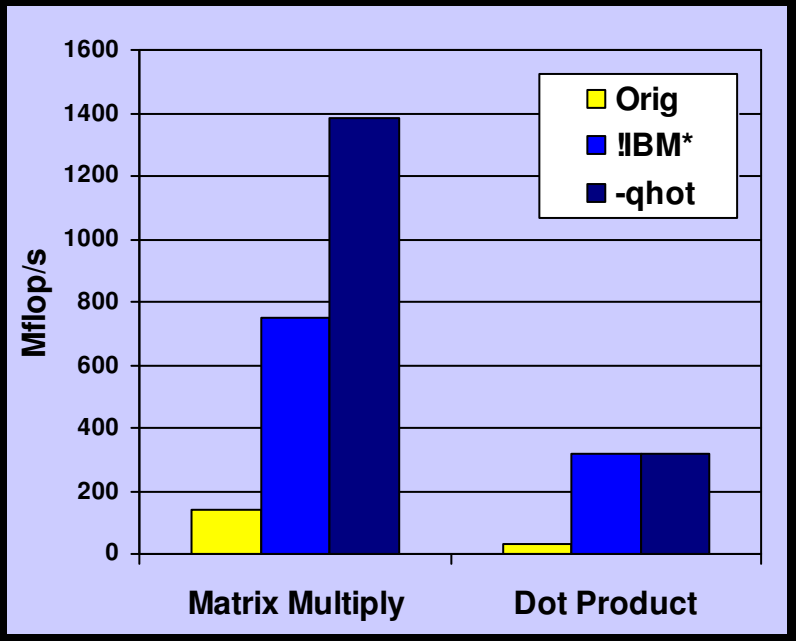


POWER4 1.3 GHz

Subscript Reorder

- Reference on second or higher subscript has bad strides
- Reduced performance
- Fixed by:
  - -qhot can interchange some loops
  - !IBM* SUBSCRIPTORDER(A(2,1))

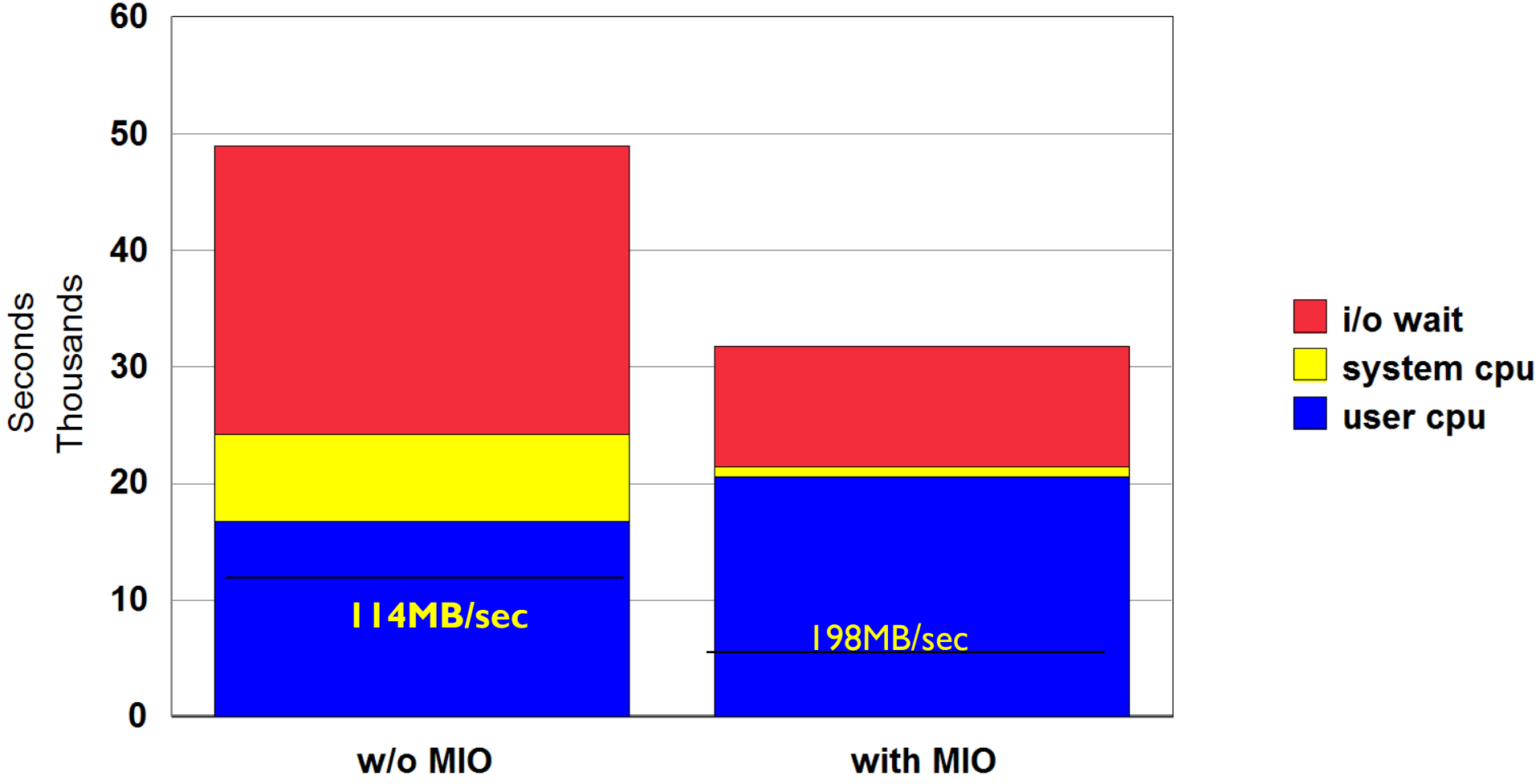
```
do i=1,n
  do j=1,m
     s = s + A(i,j)
  end do
end do
```



POWER4 1.3 GHz

MIO I/O Library

mio@us.ibm.com



16 cpu 32GB NH2 node, 8 SSA, 16 loops, 4 disk/loop,2.2M dof, 767GB I/O, 8 copies,2GB memory per copy

POWER6

Programming for POWER6

- P6 is very different architecture compared to P5
  - In-order
  - Design changes to achieve high frequency
- Potential pitfalls
  - EA Guess Wrong / Indexed Loads
  - Conditional Branches
  - Load-Hit-Store / Float-Integer conversion
  - Miscellaneous
- Options
  - Recompile for POWER6 “-qarch=pwr6 –qtune=pwr6”
  - Try different compiler options
  - Recode only if necessary

EA Guess Wrong/Indexed Loads

- A(index(i))
- Xform transform produces assembly inst.
  - ldfx fp1, EA,rA
    - EA – Base address of array
    - rA - offset
- Design of ERAT uses predictor corrector method
  - Guess is based on base address EA, if offset causes carry over in first 40bit reject occurs, stall at least 11 cycles
  - Only an issue if array is larger than 16 MB
- Fixed by new compiler
  - Explicitly compute EA + rA
  - Requires one add & extra register

Example NASPB CG

```
do j=1,lastrow-firstrow+1
    d = 0.d0
    do k=rowstr(j),rowstr(j+1)-1
      d = d + a(k)*z(colidx(k))
    enddo
    r(j) = d
  enddo
```

CG Class C (large arrays)	
System	%Improvement w/ noxform
p6	40%
P5+	-20%
CG Class B (small arrays)	
P6	-3%

Compare default options -qarch=pwr5 –qtune=balanced with –qdebug=noxform

Joseph Robichaux, Mathias Puetz: 2007

Floating Point Comparisons/Branches

- **On P6 Floating Point Compare to branch (fcmp) has 12 cycle issue-issue delay**
  - Also synchronization on conditional register (CR) forces lock on all CR registers
- **Problem for conditionals in loop**
  - Along with in-order restrictions can only evaluate one branch at a time
  - One solution is to use conditional vectorization
    - Minimize number of conditionals
    - Amortize work in conditional

```
DO i = 1,size
    dx = x(1) - y(1,i)
    …
    r2 = dx**2 + dy**2 + dz**2
    IF (r2 < cut2) THEN
      df = force(r2,qx,qy(i));
      fx(1) = fx(1) + dx*df
      …
      fy(1,i) = fy(1,i) - dx*df
      …
    END IF
  END DO
```

	Improvement relative to Hand Tuning	
Conditional Type	**P5**	**P6**
One Variant	**-20%**	**-73%**
Two Variant	**+22%**	**-53%**

Compare hand tuned vectorization with best compiler option

Joseph Robichaux, Mathias Puetz: 2007

Load Hit Store / Float-integer Conversion

- No store forwarding on P6
- Load dependent on store must wait for store to be written to L1
  - At least 11 cycle delay (likely larger)
- Float-integer conversions implemented with store/load
- Solutions
  - Move float-int conversion ahead of use
  - Use FRIZ/FRIM to deal with rounding
- Hand tuning still required

```
do i = 1,n
    …
    hinv   = 1./h
    …
    ridx  = px*hinv
    idx   = ridx
!  Original : eps   = px - idx*h
    eps   = px - FRIZ(ridx)*h
    eps1  = 1.0 - eps
    …
    wght(-1) = eps1 * eps1 * eps1 / 6.0
! Use idx, should be loaded to gpr by now
    …
    qgrid(idx-1) = qgrid(idx-1) + wght(-1) * q(i)
End do
```

Improvement relative to Hand Tuning	
P5	P6
-85%	-90%

Joseph Robichaux, Mathias Puetz: 2007

Other Operations

- Fixed point multiply
  - Fixed point multiply executed in FP unit, 17 cycles
  - Pairing FXU multiplies helps hide latency can issue two pairs in 19 cycles
- Store Queues
  - Only 16 store queues (SRQ) on P6 compared to 32 on P5
  - May have different limits for unrolling on P6
- Store Chaining
  - Stores in same cacheline can take advantage of store chaining
    - Write both to L1 & L2 in one cycle
    - Look for paired stores in assembly

Joseph Robichaux, Mathias Puetz: 2007

BLAS3 Kernel Optimization

Doug Petesch, IBM

m = (millions)
n=p = (7 to 12)

```
do k = 1, p
  do j = 1, n
    do i = 1, m
      C(i,k) = C(i,k) - A(i,j) * B(k,j)
    end do
  end do
end do
```

Equivalent to:

```
call DGEMM('n','t',m,n,p,-1.0D0,A,m,B,n,1.0D0,C,m)
```

3 Approaches

1) ddot method

```
do m1 = 1,M,2048  ! parallel
  m2 = min(m1+2048-1,M)
  do k=1,P,3      ! 3x4 unrolling
   do i=m1,m2,4
    do j=1,N      ! short loop
     t00 = t00 - B(k  ,j)*A(i  ,j)
     t10 = t10 - B(k+1,j)*A(i  ,j)
     t01 = t01 - B(k  ,j)*A(i+1,j)
     t11 = t11 - B(k+1,j)*A(i+1,j)
     t02 = t02 - B(k  ,j)*A(i+2,j)
     t12 = t12 - B(k+1,j)*A(i+2,j)
     t03 = t03 - B(k  ,j)*A(i+3,j)
     t13 = t13 - B(k+1,j)*A(i+3,j)
     t20 = t20 - B(k+2,j)*A(i  ,j)
     t21 = t21 - B(k+2,j)*A(i+1,j)
     t22 = t22 - B(k+2,j)*A(i+2,j)
     t23 = t23 - B(k+2,j)*A(i+3,j)
    end do
   end do
  end do
end do
```

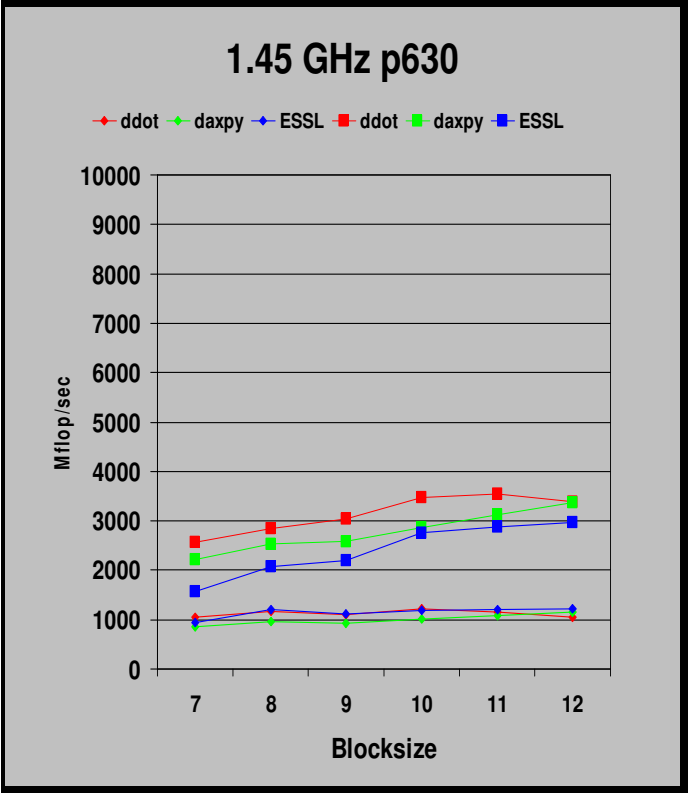
2) daxpy method

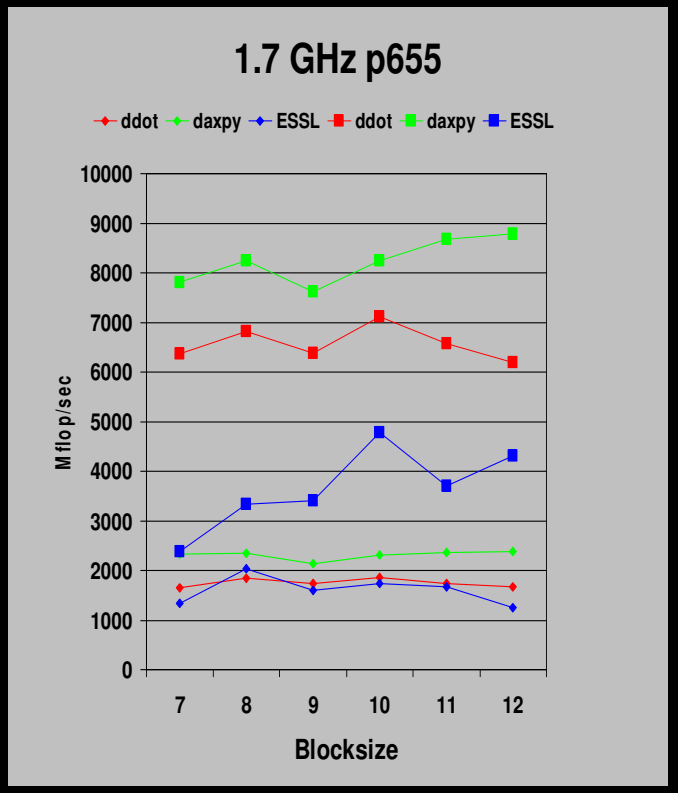
```
do m1 = 1,m,2048   ! strips that fit in L2
  m2 = min(m1+2048-1,m)
  do k = 1, p
    do j = j1, n, 4
      do i = m1, m2
        c(i,k) = c(i,k) - a(i,j) * b(k,j)
&                       - a(i,j+1) * b(k,j+1)
&                       - a(i,j+2) * b(k,j+2)
&                       - a(i,j+3) * b(k,j+3)
      end do
    end do
  end do

  end do
```

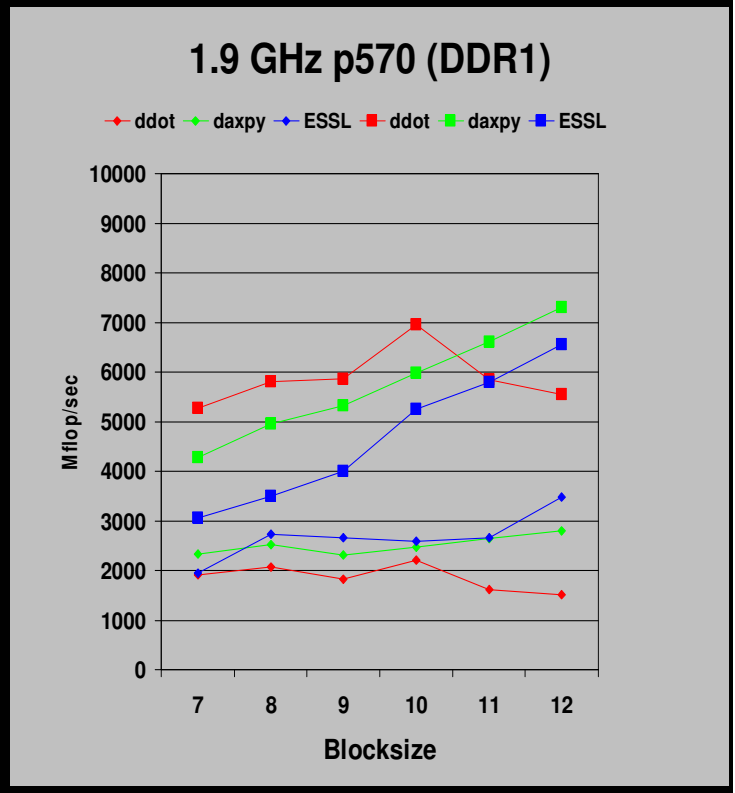
3) ESSL Call DGEMM from ESSL

Kernel Optimization

all 4-way nodes, single and 4 processor results
Mflop/sec, higher is better







November 2004

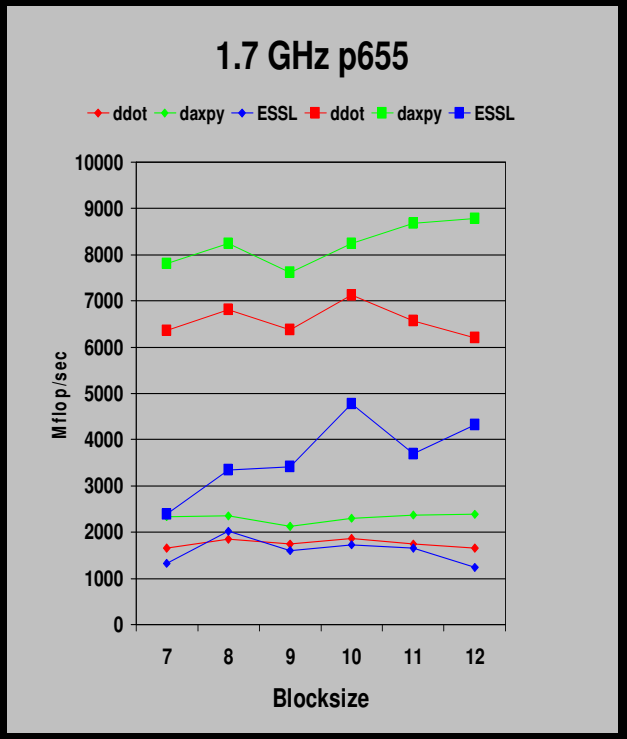
ESSL 4.1

ESSL 3.3

ESSL 4.2

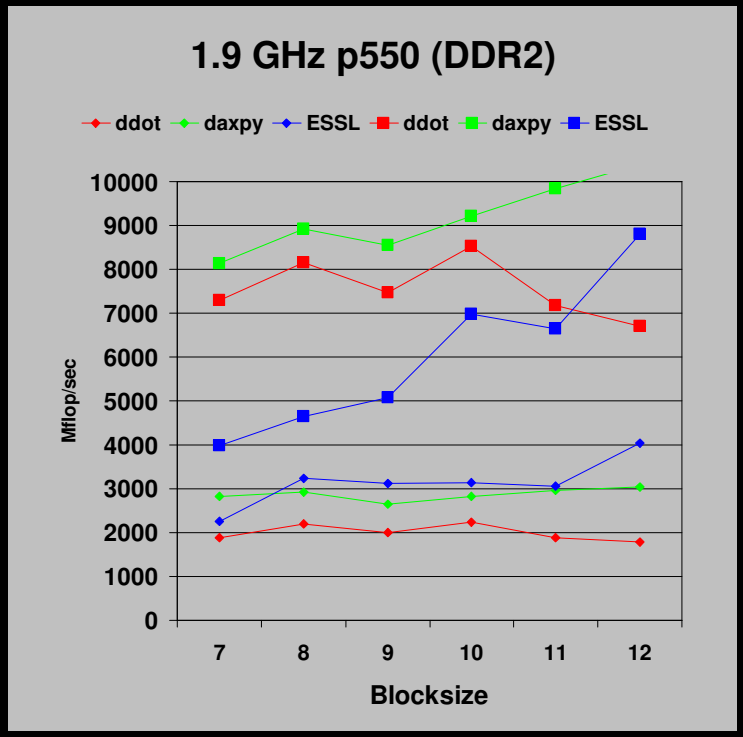
Kernel Optimization

all 4-way nodes, single and 4 processor Mflop/sec, higher is better

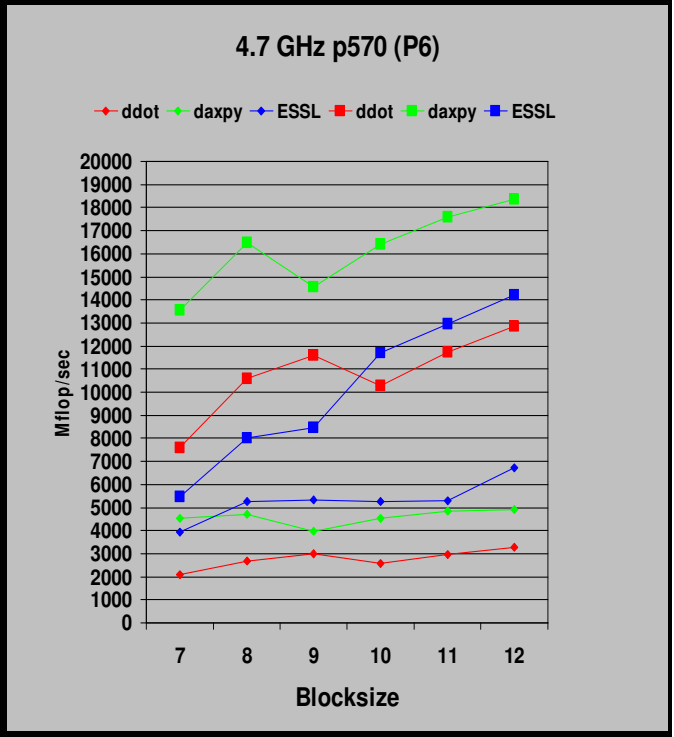


ESSL 3.3

November 2004



ESSL 4.2



ESSL 4.2

April 2007

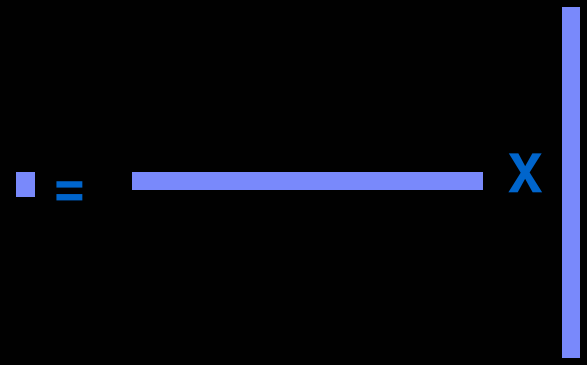
Observations

- ddot approach is best for weak memory bandwidth machines
- daxpy approach is best for stronger memory bandwidth
- ESSL version of DGEMM is not very good for long skinny matricies
- Which algorithm do you use for a single executable that will run on many different POWER machines?

Kernel Optimization:

part2

```
do i=1,N
 do k=1,L
   do j=1,M
    s(k,i) = s(k,i) + y(j,i)*x(j,k)
   end do
 end do
end do
```



- **All compiled with**
  - **"-q64 –qsmp=omp –qnosave –O3 –qnostrict –qextname"**
- **Existing binary ran poorly on POWER6**
- **Compiling with "-qarch=pwr6" was slightly slower on POWER6 than "-qarch=pwr5"**

Compiler options	POWER5+ p575 2.2GHz	POWER6 p570 4.7GHz
xlf8.1 –qarch=pwr3 –qtune=pwr5	3,380	510
xlf10.1 –qarch=pwr5 –qtune=pwr5	3,500	670
xlf11.1 –qarch=pwr5 –qtune=pwr5	4,760	7,940
xlf11.1 –qarch=pwr5 –qtune=pwr6	4,200	7,690
xlf11.1 –qarch=pwr5 –qtune=balanced	4,310	7,860

STAR-CD Performance